Journal of Parallel and Distributed Computing 126 (2019) 121–133

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Dynamic replication and migration of data objects with hot-spot and cold-spot statuses across storage data centers

Yaser Mansouri [*], Rajkumar Buyya

Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

HIGHLIGHTS

- Proposing a cost model which consists of storage, read, write, and potential migration costs.
- Optimizing cost of dynamic data replication of across Geo-distributed storage services.
- Migrating data between storage classes based on the status of data to optimize cost.

ARTICLE INFO

Article history:
Received 15 September 2017
Received in revised form 9 March 2018
Accepted 11 December 2018
Available online 21 December 2018

Keywords:
Storage data center
Data replication
Data migration
Hot-spot data
Cold-spot data

ABSTRACT

Cloud Storage Providers (CSPs) offer geographically dispersed data stores providing several storage classes with different prices. A vital problem faced by application providers is how to exploit price differences across data stores to minimize monetary cost of applications that include hot-spot objects that are accessed frequently and cold-spot objects that are often accessed far less. This monetary cost consists of replica creation, storage, Put, Get, and potential migration costs. To optimize such costs, we first propose the optimal solution that leverages dynamic and linear programming techniques with the assumption that the workload on objects is known in advance. We also propose a lightweight heuristic solution, inspired from an approximate algorithm for the Set Covering Problem, which does not make any assumption on the object workload. This solution jointly determines object replicas location, object replicas migration times, and redirection of Get (read) requests to object replicas so that the monetary cost of data storage management is optimized while the user-perceived latency is satisfied. We evaluate the effectiveness of the proposed lightweight algorithm in terms of cost savings via extensive simulations using CloudSim simulator and traces from Twitter. In addition, we have built a prototype system running over Amazon Web Service (AWS) and Microsoft Azure to evaluate the duration of objects migration within and across regions.

1. Introduction

Well known Cloud Storage Providers (CSPs) such as Amazon Web Service (AWS), Microsoft Azure, and Google offer several storage classes with different prices. The price of storage classes is different across CSPs, and it is directly proportional to the performance metrics like availability, durability, etc. For example, Reduced Redundant Storage (RRS) is an AWS's storage class that enables users to reduce their cost with lower levels of redundancy as compared to Simple Storage Service (S3).

CSPs also charge their users/application providers for network resources in different prices. They charge users for outgoing data, while the cost for ingoing data is often free. They may also charge their users at a lower cost when the data are transferred across DCs operated by the same cloud provider (e.g., AWS). This diversification of the storage and network prices plays an essential role in the optimization of the monetary cost spent on using cloud-based storage resources. This cost also affected by the expected workload of an object in online social networks (OSNs). The object might be a photo, a tweet, or even an integration of these items that share similar Get (read) and Put (write) access rate pattern. The object workload is determined by how often it is read and updated. There is a strong correlation between the object workload and the age of object, as observed in online social networks [17]. That is, the object uploaded to OSNs receives dominating more Gets and Puts during its early lifetime, and such object is in hot-spot status and is said to be network-intensive. Then the object cools over time and receives fewer and fewer Gets and Puts. Such object is in cold-spot status and is said to be storage-intensive.

Therefore, with time-varying workload and different prices of storage classes, acquiring the cheapest network and storage resources in the appropriate time of the object lifetime plays a vital

[*] Corresponding author.
E-mail address: yase@student.unimelb.edu.au (Y. Mansouri).

https://doi.org/10.1016/j.jpdc.2018.12.003
0743-7315/© 2018 Elsevier Inc. All rights reserved.

role in the cost optimization of data management in OSNs across CSPs. This cost consists of replica creation, storage, Get, Put, and potential migration costs. To optimize these costs, cloud users are required to answer the following questions: (i) which storage class from which CSP should host the object (i.e., placing), (ii) which replica should serve the specific Get (i.e., Get requests redirection), and (iii) when the object replica should probably be migrated from a storage class to another one operated by the similar or different DCs.

We previously investigated some of these questions in a dual cloud-based storage architecture that optimizes the cost of object between two DCs with two storage classes [14]. The findings in terms of cost savings obtained from this architecture motivated us to extend it across multiple data stores [15]. In [15], we proposed object placement algorithms that determine the location of limited and fixed number of object replicas with time-varying workloads. These algorithms fail to dynamically determine the number of object replicas. Moreover, they suffer from high time complexity when the object receives Gets and Puts from a wide range of DCs, and consequently demands many replicas to provision Gets and Puts within the latency constraints specified by users. To tackle these issues and answer the aforementioned questions, we propose a lightweight algorithm that demands low time complexity, thereby making it tailored for applications (e.g., OSN) that host a large number of objects.

The lightweight algorithm makes a three-fold decision for cost optimize of the data storage management: replicas location, replicas migration time from a storage class to another one operated by a single DC or two DCs, and redirection of Gets to replicas. In addition to the cost optimization, the response time of Puts and Gets is also a vital performance criterion from the perspective of users. We consider the latency constraint as a service level objective (SLO) and define it as the elapsed time between issuing a Get/Put from a data center (DC) and retrieving/writing the required object from/into the data store.

In summary, by wisely taking into account the pricing differences for storage and network resources across CSPs and time-varying workloads of objects, we are interested in reducing the cost of data storage management (i.e., replica creation, storage, Get, Put, and migration costs) so that the response time for Gets and Puts is met. To address this issue, we make the following key contributions:

- We introduce a cost model that includes replica creation, storage, Get, Put, and potential migration costs. This cost model integrates the response time for Gets and Puts in a cost optimization problem.
- We solve this optimization problem by exploiting linear and dynamic programming where the exact future workload is assumed to be known *a priori*. Due to the requirement of high time complexity, we propose a lightweight algorithm that makes key decisions on replica placement, Get requests redirection, and replicas migration time without any knowledge of the future workloads of objects.
- We conduct extensive experiments to show the effectiveness of the proposed solution in terms of cost savings by using real-world traces from Twitter [11] in the CloudSim simulator [2].
- In addition, we have built a prototype system running over AWS and Microsoft Azure cloud providers to measure the duration of objects migration within a region and across regions.

The rest of this paper proceeds as follows. Section 2 discusses the related work. Section 3 presents the system model and formally defines the optimization problem. Section 4 is devoted to the proposed solutions. Sections 5 and 6 provide experimental evaluation of the proposed lightweight solution. Finally, Section 7 concludes the paper with future directions.

2. Related work

Pricing differences within and across cloud storage services have attracted recent research attentions in cost optimization of data storage management. Here, we investigate the state-of-the-art literature in this respect.

FCFS framework [18] used two storage services (i.e., a cache and a storage class) in a single data store. In FCFS, two online algorithms have been deployed to optimize the cost of cloud file systems. This framework did not leverage pricing differences across data stores. The solution deployed in FCFS is not applicable for our cost optimization problem. This is because (i) FCFS need not to deal with latency constraints, potential migration cost, and optimizing writing cost, and (ii) it makes a decision just on when data is migrated from cache to storage and vice versa (i.e., time) while we require to make a two-fold decision: when data should be migrated to which storage class operated by a single DC or two DCs. That is, we need to make a decision on the *time* and *place*.

SPANStore [21] optimized cost by using pricing differences among CSPs while the required latency for the application is guaranteed. It used a storage class across CSPs for all objects without considering their read/write requests, and consequently it did not require to migrate objects between storage classes. Different from SPANStore, our work utilizes two storage classes to save more cost based on the objects statuses. This causes object migration between storage classes. Moreover, the time complexity of the algorithm used in SPANStore exponentially grows with the number of DCs, as compared to the quadratic growth of the proposed heuristic solution.

Cosplay [9] optimized the cost of data management across DCs – belonging to a single cloud – through swapping the roles (i.e., master and slave) of data replicas in the OSN. ES^3 [12] leveraged all pricing models to determine the reservation amount on DCs belonging to different CSPs to optimize the data management cost. It also used a genetic algorithm based method to further optimize the reservation benefit. In contrast to Cosplay and SPANStore, ES^3 explicitly leverages the tiered pricing and outperforms SPANStore in cost savings as experimentally demonstrated. Compared to our work, both Cosplay and ES^3 did not utilize two storage classes though they are orthogonal to our work for further cost optimization. Chen et al. [3] investigated the problem of placing replicas and distributing requests (issued by users) to optimize cost while meeting QoS requirements in a Content Delivery Network (CDN) utilizing cloud storage offered by a single CSP. In contrast, our work considers write requests which raise the cost of consistency as a matter.

There are several factors affecting data migration: the changes to the parameters of cloud storage (e.g.,price), optimization requirements, and data access patterns. In respect to the second factor, Qiu et al. [19] designed a dynamic control algorithm to optimally migrate data from private clouds to public ones. Wu et al. [22] focused on predicting access rate to video objects (read-only objects), and based on this observation the objects are dynamically migrated across DCs. In contrast, our study exploits pricing differences across data stores with different storage classes and dynamic migration to minimize the cost of objects that receive Gets and Puts. Puts on objects raise cost of consistency as a matter. Mseddi et al. [16] designed a scheme to create/migrate replicas across data stores with the aim of avoiding network congestion, ensuring availability, and minimizing the time of data migration. While our proposed solution optimizes monetary cost of data storage management.

In contrast to all described solutions above, our proposed algorithm exploits pricing differences across data stores (offered by different CSPs) with different storage classes. This work is in line with our previous studies [14,15] and aims at reducing the monetary cost when objects receive Gets and Puts from a wide range of DCs. This work is closely aligned with applications that host a large number of users since it uses a lightweight solution as compared to [15].

3. System model, cost model, and cost optimization problem

3.1. System model

Our system model employs different cloud providers that operate Geo-distributed DCs. DCs from different cloud providers may be co-located, but they offer several storage classes with different prices and performance metrics. Using each storage class can be determined by the user's objective (e.g., monetary cost optimization).

In our system, each user is assigned to his/her closest DC among the DCs as his/her home DC. A user creates objects (e.g., tweet or photo) and posts on his/her Twitter Feed or Facebook Timeline. The object is replicated in several DCs based on its Gets and Puts, the number of the user's friends/followers, and the required access latency to serve Gets. These replicas are named *slave* replicas, as opposed to the *master* replica stored in the home DC. The master/slave replica of the object is in *hot-spot* status if it receives many Gets and Puts, and in *cold-spot* status if it receives a few. These statuses of the object replica probably lead to the replica migration between storage classes. To do this, our system uses the *stop and copy* migration technique in which the Gets are served by the DC that the object must be migrated from (called *source DC*) and the Puts are handled by the other DC that the object must be migrated to (called *destination DC*) [20]. The unit of data migration is the *bucket* abstraction which is the same as that in Spanner [5]. The bucket consists of the objects owned by a specific user.

In the system model, a DC is referred as a *client* DC if it issues a Get/Put for an object. A DC is named as a *server* DC if it hosts a replica of an object. A DC is a *client* and *server* DC at the same time for an object if it stores a replica of the object and serves the Puts and Gets for that object.

3.2. Cost model

We assume a time-slotted system in which each slot lasts for $t \in [1 \ldots T]$. This system is represented as a set of independent DCs, D, where each DC d is associated with a tuple of four cost elements. (i) $S(d)$ denotes the storage cost per unit size per unit time (e.g., bytes per hour) in DC d. (ii) $O(d)$ defines out-network cost per unit size (e.g., byte) in DC d. (iii) $t_g(d)$ and $t_p(d)$ represent transaction cost for a bulk of Gets and Puts in DC d, respectively

Assume that a set of objects is created in time slot t. Let $r_{d_c}(t)$ and $w_{d_c}(t)$, respectively, be the number of Gets and Puts for the object with size $v(t)$ from client DC $d_c(t)$ in t. For Gets, let client $d_c(t)$ is served by server DC $d_r(t)$ that hosts a replica of the object in t. This is denoted by $d_c(t) \rightarrow d_r(t)$, which is binary, being 1 if $d_c(t)$ is served by $d_r(t)$ and being 0 otherwise. Note that it is no need for assignment of $d_c(t)$s to $d_r(t)$s for Puts since these requests issuing from $d_c(t)s$ must be submitted to all $d_r(t)$s. The number of replicas, denoted by r, for each object is variable in each time slot, and depends on the object workload, the required access latency, and the number of client DCs issuing Gets/Puts. Table 1 summarizes key notations used in this paper.

We define an objective function as to choose the placement of the object replicas ($d_r(t)$s) and to determine the assignment of client DCs $d_c(t)$ to a server DC $d_r(t)$ so that the replica creation, storage, Get, Put, and potential migration costs for the object during $t \in [1 \ldots T]$ are minimized. To find the objective function, we formally define the following costs.

Algorithm 1: The lower-bound number of replicas

```
Input : D: a set of DCs $d$, $D_c$: a set of client DCs $d_c$, latency
        between each pair of DCs, and latency constraint $L$
Output: $\lfloor r \rfloor$
1  Initialize: $\lfloor r \rfloor \leftarrow 0$
2  forall $d \in D$ do
3      forall $d_c \in D_c$ do
4          if $l(d, d_c) \leq L$ then
5              Assign DC $d_c$ to DC $d$ as a potential DC $d_p$
6          end
7      end
8  end
9  Sort DCs $d_p$ according to their assigned number of $d_c$s in
   descending order.
10 while $D_c \neq \emptyset$ do
11     Select DC $d_p$ as $d_r$ and remove its assigned $d_c$s from $D_c$ as
       well as from the set of client DCs assigned to other
       potential DCs $d_p$ which still are not selected as a server
       DC $d_r$.
12     $\lfloor r \rfloor \leftarrow \lfloor r \rfloor$+1
13 end
```

Replica Creation Cost: Once user creates an object in his/her home DC, the system may need to replicate this object in the DCs d. To do so, the system first reads the object from either the home DC d_h or the server DC d_r and then writes it into the DCs d. We refer to this cost as a *replica creation cost*, which is minimized for r replicas as below. (i) The system directly reads the object from the home DC d_h and replicates in $(r-1)$ DCs. This cost equals $v \times (r-1) \times O(d_h)$ (Fig. 1(a)). It is worth noticing that the object in the home DC is considered as a replica. (ii) The system first reads the object from the home DC d_h and replicates in the DC d, and from this DC the object is read and is replicated in $(r-2)$ DCs (Fig. 1(b)). In this case, the cost is $v \times (O(d_h) + (r-2) \times O(d))$ and is minimized by computing the cost for each DC $d \in D - \{d_h\}$ as a server DC. Therefore, the replicas creation cost is

$$min_{d \backslash d_h}[(r-1) \times O(d_h), O(d_h) + (r-2) \times min_d\, O(d)] \times v. \quad (1)$$

Since in this step, the number and the location of replicas still have not been specified, the system requires to calculate the *lower-bound number of replicas* as summarized in Algorithm 1. This algorithm assigns client DCs d_c to each DC d as DC d_p if the latency between DCs d_c and d is within the latency constraint (lines 2–8). Then, the algorithm sorts DCs d_p according to their number of assigned client DCs d_c (line 9), and finally selects the DC d_p one after another as a server DC d_r until all DCs d_c are served by a server DC d_r (lines 10–13).

Storage Cost. The storage cost of an object in time slot t is equal to the storage cost of all its replicas in DCs d_r. Thus, this cost is equal to

$$\sum_{d_r} S(d_r) \times v. \quad (2)$$

Get Cost. The Gets cost of an object in time slot t is the cost of Gets issued from all DCs d_c and the network cost for retrieving the object from DCs d_r. Hence, this cost is given by

$$\sum_{d_r} \sum_{d_c} (d_c \rightarrow d_r) \times r_{d_c} \times [t_g(d_r) + v \times O(d_r)]. \quad (3)$$

Put Cost: The Puts cost of the object in time slot t is the cost of Puts issued by all client DCs and the propagation/consistency cost to synchronize all replicas. As shown in Fig. 2(a), in the first step the client DC updates its server DC and the home DC for which

Table 1
Summary of key notations.

Symbol	Meaning
D	A set of DCs
T	Number of time slots
D_x	If "x = c", D_c is the set of client DCs. If "x = r", D_r is the set of server DCs. If "x = p", D_p is the set of potential DCs to host a replica.
d_x	If "x = c", d_c is a client DC. If "x = r", d_r is a server DC. If "x = p", d_p is a potential DC to host a replica. If "x = h", d_h is a home DC.
$S(d)$	The storage cost of DC d per unit size per unit time
$O(d)$	Out-network price of DC d per unit size
$t_g(d)$	Is transaction/request cost for a bulk of Gets in DC d
$t_p(d)$	Is transaction/request cost for a bulk of Puts in DC d
$v(t)$	The size of the object in time t
$r_{d_c}(t)$	Number of read requests from d_c in time slot t
$w_{d_c}(t)$	Number of write requests issued from d_c in time slot t
r	Number of replicas of the object
$t_m^{d_r}$	Migration time of a replica in DC d_r
$\alpha^d(t)$	A binary variable indicating whether a replica is in DC d in time slot t or not
$d_c(t) \to d_r(t)$	A binary variable, being 1 if the DC d_c is served by DC d_r and being 0 otherwise.
$C_x(.)$	If "x = R", $C_R(.)$ is the residential cost. If "x = M", $C_M(.)$ is the migration cost.
$C_B^{d_r}(t)$	Is the cost–benefit for a replica of the object in DC d_r in time slot t.
$C_L^{d_r}(t_m, t)$	Is the lost cost–benefit for a replica in DC d_r during $[t_m^{d_r}, t]$.
L	An upper bound of delay on average for Gets and Puts to receive response
$l(d_c, d_r)$	The latency between DC d_c and DC d_r

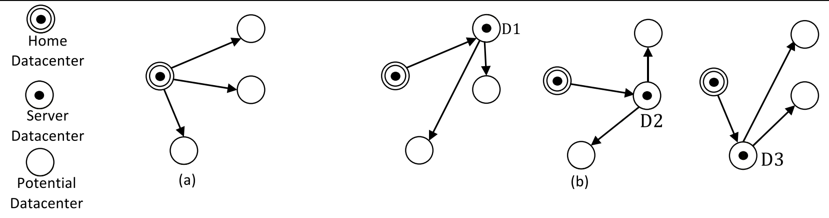


Fig. 1. Replica creation via (a) home DC and (b) potential DCs D1, D2, and D3.

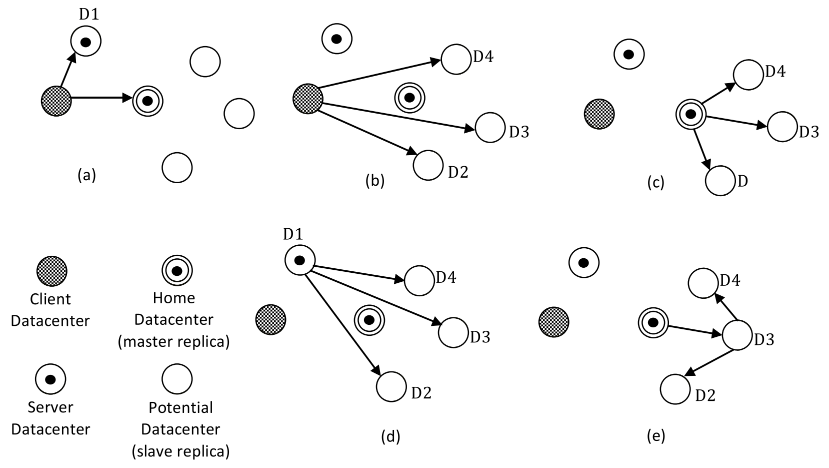


Fig. 2. Put propagation policy. (a) Client DC first updates its server DC and the home DC. DCs hosting a replica are updated via (b) the client DC, (c) the home DC, and (d) the server DC (i.e., DC D1) that serves the client DC. (e) The relayed propagation via DC D3 which is updated by the home DC.

strong consistency is guaranteed. Other replicas are then updated to guarantee eventual consistency. Thus, a DC with the minimum cost in network (either already updated or not updated) is selected as shown in Figs. 2(b)–2(e). If the selected DC has already been updated, as shown in Figs. 2(b)–2(d), then it updates other replicas in $D2$, $D3$, and $D4$. Otherwise, as depicted in Fig. 2(e), if the selected DC has not already been updated (e.g., $D3$), then it requires to be synchronized via one of the updated DCs, and then propagates data to other DCs ($D2$ and $D4$). This process is repeated for all DCs (i.e., D2, D3, and D4 in Fig. 2) not updated in the first step to exploit the discount on the network cost of those DCs operated by the same cloud providers. Formally, Puts cost is defined as

$$\sum_{d_c}[w_{d_c} \times (\begin{cases} v(O(d_c)) + t_p(d_h), & if\ d_c \in D_r(t) \\ v(2O(d_c)) + t_p(d_h) + t_p(d_c \to d_r), & o.w. \end{cases} + \min_{d_p} \sum_{d \setminus d_h \wedge d_c \to d_r} \begin{cases} v(O(d_p)) + t_p(d), & if\ d_p\ is\ d_c,\, d_h,\, or\ d_c \to d_r \\ \min_{d' \setminus d_c, d_h, d_c \to d_r} (vO(d') + t_p(d_p)) \\ +v(r-2)O(d_p + t_p(d)), & o.w. \end{cases})] \tag{4}$$

In the above Equation, the first set bracket calculates the updated cost of the home DC and the server DC that serves the client DC. The second set bracket calculates the minimum propagation cost to update $r-2$ replicas.[1]

Migration Cost: When access pattern on the object changes, the object transits from hot-spot status to cold-spot status and vice versa. This transition of the object across DCs incurs a *migration cost* which is minimized if the object migrates from $d_r \in D_r(t-1)$ with the minimum network cost to $d_r \in D_r(t)$. Thus, the migration cost for each replica in $d_r \in D_r(t-1)$ is

$$C_M^{d_r}(t-1,t) = \begin{cases} 0 & if\ d_r \in D_r(t-1) \wedge D_r(t) \\ \min_{d_r \in D_r(t-1)} O(d_r) \times v(t-1) & o.w. \end{cases} \tag{5}$$

As seen in Eqn. (5), if the placement of the replica in t and $t-1$ is the same, then migration cost is zero. Otherwise, the replica migrates from DC $d_r \in D_r(t-1)$ to $d_r \in D_r(t)$, and application providers incur a migration cost and make a *cost–benefit*. The cost–benefit is the difference between the residential cost of the replica in the old location $d_r \in D_r(t-1)$ and the one in the new location $d_r \in D_r(t)$. The *cost–benefit* obtained from replica migration is very important for users to make a decision on whether to migrate the replica or not. Algorithm 2 summarizes a wise decision in this respect.

For ease of algorithm explanation, we introduce three notations. Assume that $t_m^{d_r}$ denotes the last time of migration for the replica object in DC d_r. Also suppose $C_B^{d_r}(t)$ is a cost–benefit for DC d_r in time t and $C_L^{d_r}(t_m,t)$ is the summation of cost–benefits which have been lost for the replica in DC d_r during period $[t_m^{d_r}, t]$. This happens when $C_L^{d_r}(t_m,t)$ cannot compensate the migration cost of replica from the old location in $t_m^{d_r}$ to the new one in t.

Algorithm 2 excludes the object replicas which have the same location in $t-1$ and t since these replicas do not require migration (line 1). For those replicas having potential to be migrated, the algorithm first calculates the total of *migration* cost and *residential* cost of the replica stored in $d_r \in D_r(t)$, i.e., $C(d_r,t)$– (line 3). Then, the algorithm computes the residential cost of the replica as if it is stored in each $d_r' \in D_r(t-1)$, i.e., $C_R(.)$– (line 5).

According to these two values, $C(d_r,t)$ and $C_R(.)$, cost–benefit $C_B^{d_r'}(t)$ is calculated in t (line 6). Now Algorithm 2 makes decision on the replica migration in the following cases. **Case 1**: If the summation of lost cost–benefit during $[t_m, t-1]$ and the cost–benefit in t is more than the migration cost (i.e. line 8), then it is cost effective to migrate the replica from the old location in $t-1$ ($d_r' \in D_r(t-1)$) to the new one in t ($d_r \in D_r(t)$)–(line 6). In this case, the DC d_r is added to $D_r^*(t)$, and the DC d_r' is removed from $D_r(t-1)$ to avoid comparison with the next DCs which may host a migrated replica (lines 9–10). Also, $C_L^{d_r}(t_m,t)$ is set to zero and the migration

Algorithm 2: Optimization of replicas migration

```
Input : $D_r(t-1)$, $D_r(t)$, latency between each pair of DCs,
        and latency constraint $L$
Output: $D_r^*(t)$: the optimized location of replicas in $t$
1  Find the intersection set of $D_r(t-1)$ and $D_r(t)$ sets and
   remove DCs in the intersection set from both $D_r(t-1)$ and
   $D_r(t)$.
2  forall $d_r \in D_r(t)$ do
3      $C(d_r,t) \leftarrow$ Calculate residential cost according to Eqns.
       (1) - (4) and the migration cost based on Eqn. (5) .
4      forall $d_r' \in D_r(t-1)$ do
5          $C_R(d_r'(t-1), d_c(t) \to d_r'(t-1)) \leftarrow$Assume as if all
           $d_c$s assigned to $d_r$ are served by $d_r'$ subject to
           $l(d_c, d_r') \le L$ and calculate the residential cost based
           on Eqns. (1) - (4) .
6          $C_B^{d_r'}(t) \leftarrow [C_R(d_r'(t-1), d_c(t) \to d_r'(t-1)) - C(d_r,t)]$
7          /*Migration happens*/
8          if $C_L^{d_r}(t_m, t-1) + C_B^{d_r}(t) \ge C_M^{d_r}(t-1,t)$ then
9              $D_r^*(t) \leftarrow D_r^*(t) \cup d_r$
10             $D_r(t-1) \leftarrow D_r(t-1) - \{d_r'\}$
11             $C_L^{d_r}(t_m,t) \leftarrow 0$, $t_m^{d_r} \leftarrow t$
12             break.
13         end
14     end
15     /*Migration does not happen*/
16     if $d_r \notin D_r^*(t)$ then
17         Find $min_{d_r'} C(d_r',t)$
18         $D_r^*(t) \leftarrow D_r^*(t) \cup d_r'$
19         $D_r(t-1) \leftarrow D_r(t-1) - \{d_r'\}$
20         $C_L^{d_r}(t_m,t) \leftarrow C_L^{d_r}(t_m,t-1) + [C_R(d_r'(t-1), d_c(t) \to$
           $d_r'(t-1)) - C(d_r,t)]$
21     end
22 end
```

time of replica in the DC d_r is updated to the current time t in order to evaluate the next potential migration time for this replica (line 11). **Case 2**: Otherwise, if the cost–benefit during period $[t_m, t]$ is less than the migration cost of replica from the DC d_r' in t_m to the DC d_r in t, then the replica migration does not happen (line 16). Thus, Algorithm 2 finds the DC d_r' with the minimum residential cost (i.e., $min_{d_r'} C_r(d_r', t)$), and then it adds the DC $d_r' \in D_r(t-1)$ to $D_r^*(t)$ and removes it from $D_r(t-1)$ (lines 17–19). Also, it adds the lost cost–benefit in t to the summation of those during period $[t_m, t-1]$ (line 20).

To show how this algorithm works, we will give a simple example as shown in Fig. 3. In this example, the set of DCs that hosts replicas in time $t-1$ are D1, D2, D3, D5, and D6. Assume the set of potential DCs can store replicas in time t are D1, D4, D7, D8, and D6. Since D1 (as home DC) and D6 are in both sets, they are excluded from replica migration process. Thus, Algorithm 2 should make a decision on whether to migrate replicas to new DCs D4, D7, and D8 in time t.

As shown in Fig. 3(b), Algorithm 2 calculates residential and migration costs of the replicas in new DCs. For example these values for D8 are 4 and 2 respectively. Then, it calculates the residential cost of the replica as if it stays in the old DCs D2, D3, and D5, as represented in parenthesis. Now the cost of the replica in D4 is compared to that in D2, D3, and D5. Since the cost of the replica in D4 (3+2) is equal to the residential cost of the replica in D3 (5), the replica remains in D3 (marked with asterisk), and D4 is excluded for the next decision on the replica placement. For the next decision, as shown in Fig. 3(c), the residential cost of the

[1] Some cloud providers offer discount on the network cost if data is transferred among their DCs. For simplicity, this discount is not formulated in the Put and Migration costs, though it is considered in the performance evaluation in Section 5.

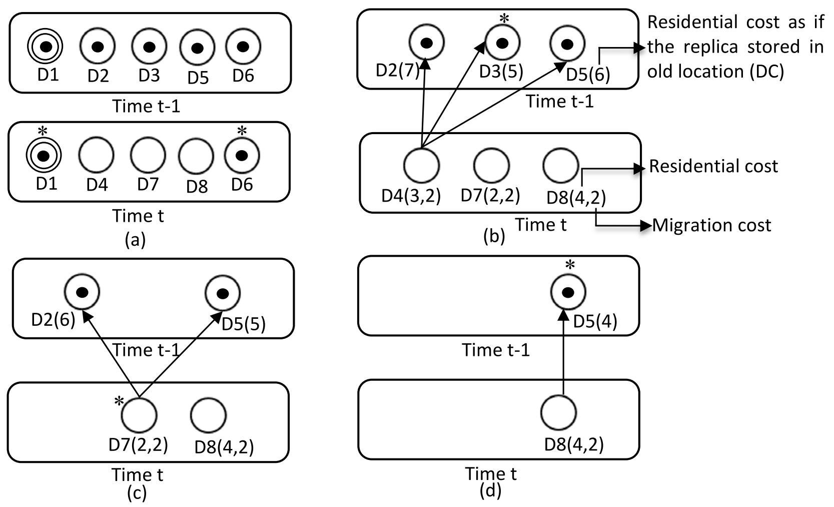


Fig. 3. An example of illustrating the replica migration between two consecutive time slots.

replica is calculated as if it is stored in old DCs D2 and D5. Since the residential and migration costs of the replica in D7 (2+2) is less than the residential cost in D2 (6) and D5 (5), D7 is selected to host another object replica. Thus, D7 and D2 are excluded for the next decision. In the same way, D5 is selected to host the last replica of the object. Thus, two replicas remain in the old DCs D3 and D5, and one replica migrates to the new DC D7.

The above discussed strategy uses the *stop and copy* technique [20] deployed by the single cloud system such as HBase[2] and ElasTraS [6], and in Geo-replicated systems [20]. As we desire to minimize the monetary cost of migration, we use this technique in which the amount of data moved is minimal as compared to other techniques leveraged for live migration at shared process level of abstraction.[3] We believe that this technique does not affect our system performance since the duration of migration for transferring a bucket (at most 50 MB, the same as in Spanner [5]) among DCs is considerably low as shown in Section 6.

Therefore, with the defined residential and migration costs based on Eqs. (1)–(5), the total cost of the object replicas in t is defined as

$$C_R(d_r(t), d_c(t) \rightarrow d_r(t)) + C_M^{d_r}(t-1, t), \tag{6}$$

where $C_R(.)$ as a *residential cost* is the summation of costs in Eqs. (1)–(4).

Besides the total cost optimization, our system respects the latency Service Level Objective (SLO) for Gets and Puts. The latency for a Get or Put is considered the time taken from when a user issues a Get or Put from the DC d_c to when he/she gets a response from the DC d_r that hosts the object. The latency between d_c and d_r is denoted by $l(d_c, d_r)$ and is evaluated based on the round trip times (RTT) between these two DCs since the size of objects is typically small (e.g., tweets, photos, small text file), and thus data transitions are dominated by the propagation delays, not by the bandwidth between the two DCs. For applications with large objects, the measured $l(d_c, d_r)$ values capture the impact of bandwidth and data size as well. This performance criterion is integrated in the cost optimization problem discussed in the following subsection.

3.3. Cost optimization problem

Given the defined system and cost models, we are required to determine the location of object replicas (d_r) and the client DCs (issuing Gets and Puts) assignment to a server DC ($d_c(t) \rightarrow d_r(t)$) so that the overall cost for the object replicas (Eqn. (6)) during $t \in [1 \ldots T]$ is minimized. This is defined as the following *cost optimization problem*:

$$\min_{\substack{d_r(t) \\ d_c(t)\rightarrow d_r(t)}} \sum_t C_R(.) + C_M(.) \tag{7}$$

s.t. (repeated for $\forall t \in [1 \ldots T]$)
(a) $\sum_{d_c} d_c(t) \rightarrow d_r(t) = 1, \qquad \forall d_r \in D_r(t)$
(b) $\sum_{d_r} d_c(t) \rightarrow d_r(t) = |D_c(t)|, \qquad \forall d_c \in D_c(t)$
(c) $\frac{\sum_{d_r}\sum_{d_c} r_{d_c(t)} \times l(d_c(t), d_r(t))}{\sum_{d_c} r_{d_c(t)}} \leq L, \qquad \forall d_c \in D_c(t), d_r \in D_r(t)$
(d) $\sum_{d_r} d_c(t) \rightarrow d_r(t) = |D_r(t)|, \qquad \forall d_c \in D_c(t), Puts$
(e) $l(d_c(t), d_c(t) \rightarrow d_r(t)) \leq L \qquad \forall d_c \in D_c(t), d_r \in D_r(t)$.
To optimally solve the above problem, we are required to replace $d_r(t)$ with $\alpha^d(t)$ that is associated to DC d. The introduced variable $\alpha^d(t)$ is binary, being 1 if the DC $d \in D$ hosts a replica of the object, otherwise 0. Thus, we apply a constraint as below:
(f) $\sum_{d \in D} \alpha^d(t) \geq 1, \qquad \forall d \in D$.

In this cost optimization problem, $C_R(.) + C_M(.)$ is calculated based on Eqn. (6), and L is as the upper bound of delay for Gets and Puts on average to receive response. The value of L is defined by users as their SLO. To reflect the real-world practicality, we consider the following constraints. Constraint (a) ensures that a single server DC d_r for every client DC d_c. Constraint (b) guarantees all client DCs are served. Constraint (c) enforces the average response time of Gets in range of L. Constraints (d) and (e) indicate that the Puts are received by all server DCs in the average response time L. Constraint (f) ensures at least a replica of the object is stored in DCs at any time.

4. Cost optimization solution

We first provide a brief demonstration of the optimal solution for the cost optimization problem, and then propose a heuristic solution to perform well in practice.

[2] Apache HBase. https://hbase.apache.org/book.html.

[3] In cloud-based transactional database, techniques such as *stop and copy*, *iterative state replication*, and *flush and migrate* in the process level are used. The interested readers are referred to [8] and [7].

4.1. Optimal solution

To optimally solve the cost optimization problem, we should find the values of $\alpha^d(t)$s and $d_c(t) \rightarrow d_r(t)$s, the same as that discussed in [15]. The value of $d_c(t) \rightarrow d_r(t)$s can be determined via linear programming once the value of $\alpha^d(t)$s is fixed. To enumerate value of all $\alpha^d(t)$s, we need to enumerate all $r-$combinations of a given set of DCs (i.e., $\binom{|D|}{r}$). Since the value of r can be ranged between 1 and $|D|$, the number of combination of $\alpha^d(t)$s is $\sum_{r=1}^{|D|}\binom{|D|}{r} = 2^{|D|} - 1$.[4]

To find the optimal solution, we use a dynamic algorithm the same as one in [15], which only differs in the number of combinations of $\alpha^d(t)$s (i.e., the combinations of DCs with r $(1 \leq r \leq |D|)$ replicas. Thus, the algorithm calculates the cost for $(2^{|D|} - 1)^2$ combinations of DCs for each time slot $t \in [1 \ldots T]$. This calculation takes time complexity of $O((2^{|D|} - 1)^2 TT_{lp})$, where T_{lp} is the required time to solve the linear programming for finding the value of $d_c(t) \rightarrow d_r(t)$s. As the optimal solution is computationally intractable, we seek a practical heuristic solution.

4.2. Heuristic solution

We first propose Replica Placement based on Covered Load Volume (RPCLV) Algorithm. This algorithm makes the iterative decision on the assignment of client DCs to the potential DCs which are within the latency constraint L. Then, by using the RPCLV algorithm, Algorithms 1 and 2, we propose the heuristic solution summarized in Algorithm 4.

4.2.1. Replica Placement based on Covered Load Volume (RPCLV) Algorithm

The RPCLV algorithm is inspired from an approximation algorithm for the Set Covering Problem [4]. This algorithm stores a replica of the object in the potential DC which has the minimum proportion of the residential cost (Eqns. (1)–(4)) and potential migration cost (Eqn. (5)) to the volume data (in bytes) read from and written into the potential DC by the client DCs. Clearly, these client DCs are within the latency constraint L of the potential DC, but they are not assigned yet. The algorithm selects this potential DC as a server DC and finds the next best potential DC to host a replica. This process is repeated until all client DCs are assigned to a potential DC as a server DC. The details are given in Algorithm 3.

The RPCLV algorithm first assigns client DCs to each potential DC $dp \in D$ if the latency between the client DC d_c and the potential DC d_p is within the latency constraint L (lines 3–10). Then, it calculates $P_{CV}(d_p)$ as the proportion of the residential and migration costs of the replica stored in the DC d_p to the total data read and written by the set of client DCs assigned to d_p (i.e., $D_c^{d_p}$)– lines 14–16. After that, RPCLV repeats the following process until every client DC is assigned to a server DC. (i) RPCLV selects d_p as d_r with the minimum value of P_{CV} for storing a replica, adds d_r to D_r, and removes d_p from D_p (lines 17–18). (ii) It also removes all client DCs assigned to this d_p from the client DCs (i.e., D_c) as well as removes them from those client DCs covered by the DC $d_{p'}$ for every $p' \neq p$ (line 19). Finally, RPCLV revises the replicas creation cost since in RPCLV the replica creation cost is calculated based on the lower-bound number of replicas (i.e., $\lfloor r \rfloor$) specified by Algorithm 1. Thus, RPCLV adds cost $|(|D_r| - \lfloor r \rfloor) \times min_{d_r \in D_r} O(d_r)|$ to the replicas creation cost if $|D_r| > \lfloor r \rfloor$; Otherwise if $|D_r| < \lfloor r \rfloor$, it subtracts this value from the replicas creation cost (line 22). Clearly, if $|D_r| = \lfloor r \rfloor$ there is no need to change the replicas creation cost.

[4] Note that neither Algorithm 1 nor Algorithm 2 is used in optimal solution since this solution enumerates all values of r as well as all combinations of object migration between $d_r \in D_r(t-1)$ and $d_r \in D_r(t)$.

Algorithm 3: Replica Placement based on Covered Load Volume (RPCLV)

```
Input : $D_c$, latency between each pair of DCs, and latency constraint $L$
Output: $D_r$: the location of replicas in $t$
1  Initialize: $D_r \leftarrow \emptyset$
2  /*Assign feasible client DCs to the potential DCs. */
3  forall $d \in D$ do
4      forall $d_c \in D_c$ do
5          if $l(d_c, d) \leq L$ then
6              Consider the DC $d$ as a potential DC $d_p$ which can host a replica of the object
7              $D_c^{dp} \leftarrow$ assign $d_c$ to the DC $dp$
8          end
9      end
10     $D_p \leftarrow D_p \cup d_p$
11 end
12 /*Assign client DCs to the potential DCs based on the cost–volume ratio until all client DCs are covered. */
13 while $D_c \neq \emptyset$ do
14     forall $d_p \in D_p$ do
15         $P_{CV}(d_p) = \frac{\sum_{d_c \rightarrow d_p} C_R(d_c, d_c \rightarrow d_p) + C_M^{d_p}(t-1,t)}{\sum_{d_c \rightarrow d_p}(r_{d_c} + w_{d_c}) \times v}$
16     end
17     Find $min_{d_p} P_{CV}(d_p)$ and store a replica of the object in $d_p$ as $d_r$
18     $D_r \leftarrow D_r \cup d_r, D_p \leftarrow D_p - d_p$
19     $D_c \leftarrow D_c - D_c^{d_p}, D_c^{d_{p'}} \leftarrow D_c^{d_{p'}} - D_c^{d_p}$
20 end
21 /*revise the replica creation cost */
22 Add $|(|D_r| - \lfloor r \rfloor) \times min_{d_r \in D_r} O(d_r)|$ to replicas creation cost if ($|D_r| > \lfloor r \rfloor$) and subtract it if ($|D_r| < \lfloor r \rfloor$).
```

Algorithm 4: The Cost Optimization Algorithm

```
Input : $T$.
Output: $C_{ove}$: The overall cost
1  Initialize: $C_{ove} \leftarrow 0$
2  $t \leftarrow 1$
3  Call the RPCLV algorithm and determine $D_r(t)$ as well as $D_c^{d_r}$ for each $d_r \in D_r(t)$
4  $C_{ove} \leftarrow$ Calculate the residential Cost $C_R(.)$ based on Eqns. (1)-(4)
5  for $t \leftarrow 2$ to $T$ do
6      Call the RPCLV algorithm and determine $D_r(t)$ as well as $D_c^{d_r}$ for each $d_r \in D_r(t)$
7      /*$C_R(.)$ and $C_M(.)$ are calculated based on Eqn. (6)*/
8      if $D_r(t-1) \neq D_r(t)$ then
9          Call the optimization of replicas migration algorithm (Algorithm 2)
10         $C_{ove} \leftarrow C_{ove} + C_R(.) + C_M(.)$
11     else
12         $C_{ove} \leftarrow C_{ove} + C_R(.)$
13     end
14 end
```

4.2.2. Cost optimization algorithm

Cost Optimization: Algorithm 4 gives the pseudocode of the proposed heuristic solution which is composed of Algorithms 1, 2, and 3. In Algorithm 4, first RPCLV is invoked to determine the locations of replicas (i.e., $d_r(t)$) as well as the assigned client DCs to

Table 2
The time complexity of Algorithms 1–3.

Algorithm				Total time complexity
Algorithm 1	O($\|D\|^2$) + Lines (2–8)	O($\|D\| log \|D\|$)+ Line (9)	O($\|D\|$) Lines (10–14)	=O($\|D\|^2$)
Algorithm 2	O($\|D\|$) + Line (1)	O($\|D\|^2$) Lines (2–18)		=O($\|D\|^2$)
Algorithm 3	O($\|D\|^2$)+ Lines (3–11)	O($\|D\|^2$)+ Lines (13–21)	O($\|D\|$) Line (23)	=O($\|D\|^2$)

server DCs (i.e., $d_c(t) \rightarrow d_r(t)$). According to the values of $d_r(t)$ and $d_c(t) \rightarrow d_r(t)$, first the residential cost of the replicas (i.e., $C_R(.)$) based on Eqns. (1)–(4) is calculated for time slot $t = 1$ (lines 2–4). Then, for each time slot $t \in [2 \ldots T]$, similarly Algorithm 4 finds the replicas location and the assignment of client DCs to server DCs (line 6) and checks whether the location of replicas in $t-1$ and t are different or not. If they are different (i.e., $D_r(t-1) \neq D_r(t)$), Algorithm 2 is called to determine which replicas of the object can be migrated and then the residential and migration costs are calculated based on Eqn. (6) (lines 8–10). Otherwise (i.e., $D_r(t-1) = D_r(t)$), the migration of replicas does not happen and only the residential cost of replicas is calculated (line 12).

Metadata: At the beginning of each time slot, the metadata is updated according to the replication policy dictated by Algorithm 4. The metadata is stored in all client DCs for a particular object in the form of a table consisting of (object id $\rightarrow$ server DCs) mapping, (server DC $\rightarrow$ client DC(s)) mapping, and how the put operations are conducted on the replicas of the object. The former mapping shows the replicas location for a particular object and the later one demonstrates which server DC serves the client DC issuing Get request for this object.

Time Complexity: Table 2 summarizes the time complexity of the heuristic solution (Algorithm 4). Algorithm 4 invokes Algorithm 3 which takes O($|D|^2$)-line 3. Then, for each $t \in [2 \ldots T]$, it invokes RPCLV with the time complexity of O($|D|^2$), and also runs Algorithm 2 with the time complexity of O($|D|^2$) if the replicas migration happens. Thus, the heuristic solution yields the time complexity of $O(|D|^2 T)$.

5. Performance evaluation

We evaluate the proposed solution for replicas placement of the objects across Geo-distributed data stores with two storage classes. Our evaluations explore three key questions: (i) How significant is our solution in the cost saving?, (ii) how sensitive is our solution to different parameters settings which are likely to have effect on the cost saving?, and (iii) How much time is required to migrate objects within and across regions? We explore the first two questions via trace-driven simulations using the CloudSim discrete event simulator [2] and the Twitter workload [11]. Simulation studies enable us to evaluate our solution on a large scale (thousands of objects). We answer the last question via the implementation of our proposed solution on Amazon and Microsoft Azure cloud providers in Section 6. This implementation allows us to measure the latencies that are required for data migration within and across regions in a real test-bed.

5.1. Experimental setting

We use the following setup for DC specifications, objects workload, users location, and experiment parameters setting.

DCs specifications: We model 18 DCs in CloudSim Toolkit [36], and among these DCs, nine are modeled according to Amazon and nine according to Microsoft Azure in different regions: 7 in America, 4 Europe, and 7 in Asia Pacific. We use two storage classes from each cloud providers: S3 and RRS from Amazon and ZRS and LRS from Microsoft Azure. S3 and ZRS host objects with hot-spot status and the remaining storage classes store objects with cold-spot status. The price of these storage classes and network services are set for each DC based on AWS and Microsoft Azure as of November 2016.[5]

Objects workload: We use Twitter traces [11] which includes users profile, a user friendship graph, and tweet objects sent by users over a 5-year period. We focus on tweet objects posted by the users and their friends on their timeline, and obtain the number of tweets (i.e., number of Puts) from the dataset. Since the dataset does not contain information of accessing the tweets (i.e., number of Gets), we set a Get/Put ratio of 30:1, where the pattern of Gets on the tweets follows Longtail distribution [1]. This pattern mimics the transition status of the object from hot- to cold-spot status. The size of each tweet object varies from 1 KB to 100 KB in the trace.

Users location: We assign each user to a DC as his/her home DC based on the following policies. (i) Closest-based policy: with the help of Google Maps Geocoding API,[6] we convert the locations of users in their profiles to Geo-coordinates (i.e., latitude and longitude). Then, according to the coordination of users and DCs, we assigned users to the nearest DC based on their locations. In the case of two (or more) DCs with the same distance from the user, one of these DCs is randomly selected as the home DC for the user. (ii) Network-based policy: users are directed to the cheapest DC in the network cost. (iii) Storage-based policy: users are mapped to the cheapest DC in the storage cost. In the last two policies, the selected DC must be within the SLO defined by users, and in the case of two (or more) DCs with the same cost in either network or storage, one of the closest DC is selected as the same way used in the first policy. We use closest-based policy to assign friends of the user to a DC. The user's friends are derived from the friendship graph of dataset.

Experiment parameters setting: We measured inter-DCs latency (18*18 pairs) over several hours using instances deployed on all 18 DCs. We run Ping operations for this purpose, and used the medium latency values as the input for our experiments. We consider two SLOs for the values of Get and Put latencies: 100 ms and 250 ms. Recall that the Put latency is the latency between the client DC to the server DC that serves it. The stored data in the system depends on the size of tweet objects, the number of friends of the users and the rate of Get (write) [14]. To understand the effects of the total stored data size in data stores on the cost performance, we define "quantile volume" parameter. This parameter with the value of "x" means that all data stores only store "x" percent of the generated total data size. We use one-month (Dec. 2010) of Tweeter traces with more than 46 K users, posting tweet on their timeline, for our experiments conducted over a 60-day period.

[5] Amazon S3 storage and data transfer pricing. https://aws.amazon.com/s3/pricing/ Azure storage pricing. https://azure.microsoft.com/en-us/pricing/details/storage/ Azure data transfer pricing. https://azure.microsoft.com/en-us/pricing/details/data-transfers/.

[6] The Google maps geocoding API. https://developers.google.com/maps/documentation/geocoding/intro.

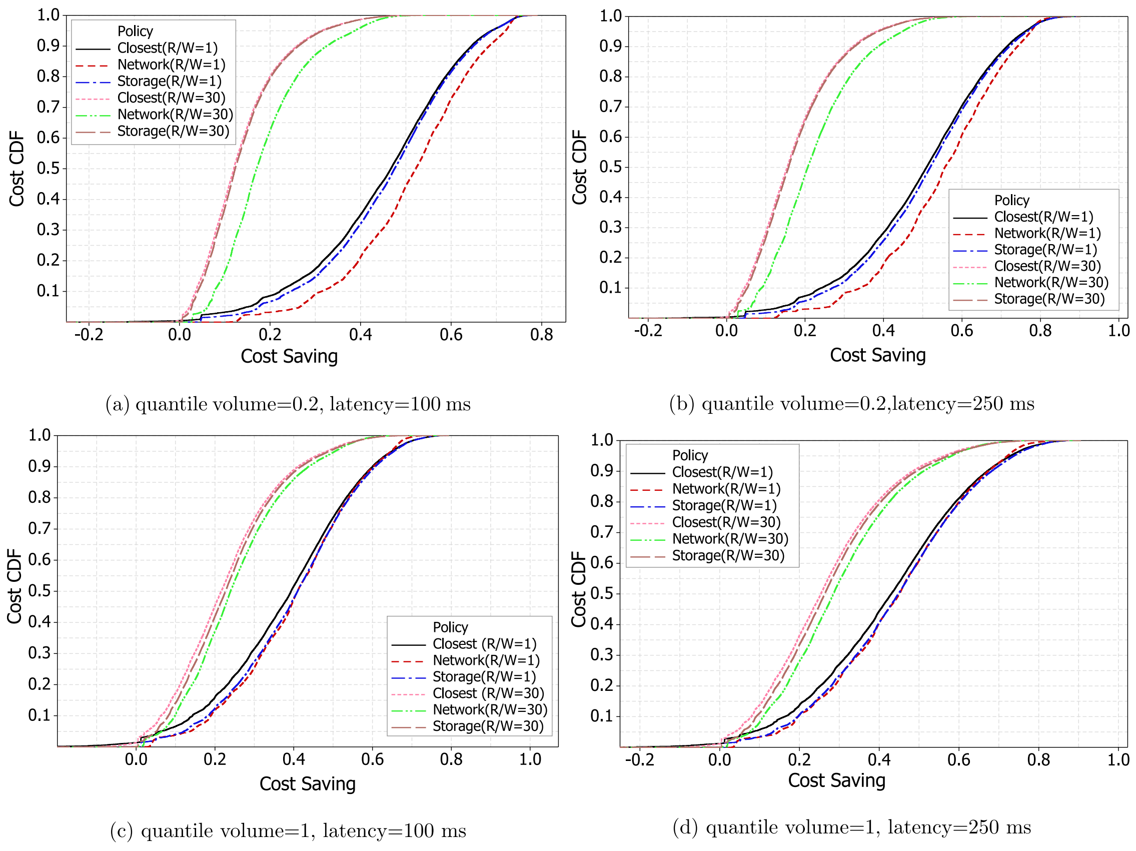


Fig. 4. Cost CDF vs. Cost saving of closest-, network-, and storage-based polices under tight (100 ms) and loose (250 ms) latency.

5.2. Results

We compare the cost savings gained by the proposed heuristic solution with the following benchmark algorithm. We also investigate the effects of parameters as their values are changed.

Benchmark Algorithm and the Range of Parameters: This algorithm permanently stores the objects in the home DC. It also replicates a replica of objects in the client DCs, upon issuing Get requests, via the home DC. The replica is stored in the client DCs until they receive Gets and Puts, and thus the replica is not allowed to migrate to another DC. This algorithm, though simple, is effective to measure the cost performance of using data stores with two storage classes offered by different DCs. In all experiments, we normalize the incurred cost of the heuristic solution to the cost of the benchmark algorithm by varying parameters in Table 3. Each parameter has a default value and a range of values for studying their impact of the parameter variations on the cost performance.

Cost performance: We present simulation results in Fig. 4, where the CDF of the normalized cost savings is grouped by the default value of quantile volume and latency. From the results, we observe the following remarks. First, there is a hierarchy between policies in the cost savings, where the network-based policy outperforms the storage-based policy, which in turn outweighs the closest-policy. This is because that the network-based policy allows users to select DC which is cheaper than their closest DC. When we go deep into the cost savings obtained from each individual DC, we realized that users in California select cheaper DC within their specified SLO instead of Amazon'DC in California. Second, all policies provide cost savings for write-intensive objects (R/W = 1) higher than read-intensive objects (R/W = 30). For example, as shown in Figs. 4(a) and 4(b), all policies cut costs up to 50% for all read-intensive objects, while they save similar costs for half of write-intensive objects and between 50%–80% for another half. Third, all policies cut costs for almost all objects, apart from 2%–3% of the objects that incur slightly more costs than as if they were replicated in each client DCs according to the benchmark algorithm.

Table 4 summarizes the average cost saving for each group of default values of quantile volume, R/W, and latency. We can see that the network-based policy achieves the highest cost saving, while the closest-based policy obtains the lowest, where the difference between these two policies is at most 6%. From the highest cost saving on average obtained by the network policy for each group of default parameters and the corresponding results in Fig. 4, we realize the following facts. As shown in Fig. 4(a), all policies save costs more than the highest cost saving on average for at least 45% (25%) of write-intensive (read-intensive) objects. As the quantile volume increases from 0.2 to 1, all policies save costs more than the highest average cost saving for 50% of read- and write-intensive objects (Fig. 4(c)). Likewise, results remain the same, or even more improvements are experienced, for objects under loose latency (Figs. 4(b) and 4(d)). Thus, we can say that all policies cut costs for most of objects more than the highest cost saving on the average obtained from the network-based policy.

We now investigate the effect of different parameters on the cost saving. For the sake of brevity, from hereafter, we report the results only for default values of parameters. We consider "closest" as the default policy since users often are mapped to the closest DC.

Fig. 5 shows the effect of quantile volume by varying it from 0.2 to 1 with the step size of 0.2 on the cost savings. As the quantile volume increases from 0.2 to 1, cost savings slightly decrease by about 6%–7% for write-intensive objects under both latency constraints. The rational is that when quantile volume = 0.2, the cost is dominated by write cost and our solution can optimize more cost. As the quantile volume increases to 1, the effect of this domination reduces. In contrast, cost savings increase by about 9% for read-intensive objects under both tight (100 ms) and loose (250 ms)

Table 3
Summary of simulation parameters.

	Policy	Quantile Volume	SLO Latency (ms)	Read to Write Ratio
Default	Closest-based	0.2,1	100, 250	1,30
Range	Closest-, network-, Storage-based	0.2–1	50–250	1–30

Table 4
Average cost savings normalized to the benchmark algorithm cost.

Quantile volume	Policy	Latency = 100 ms		Latency = 250 ms	
		R/W = 1	R/W = 30	R/W = 1	R/W = 30
0.2	Closest	44.78%	13.61%	49.21%	17.12%
	Network-based	50.82%	18.78%	54.31%	22.32%
	Storage-based	46.00%	13.98%	50.48%	17.51%
1	Closest	38.00%	22.79%	42.05%	26.80
	Network-based	40.11%	25.43%	44.06%	29.89%
	Storage-based	39.91%	23.88%	44.08%	27.96%

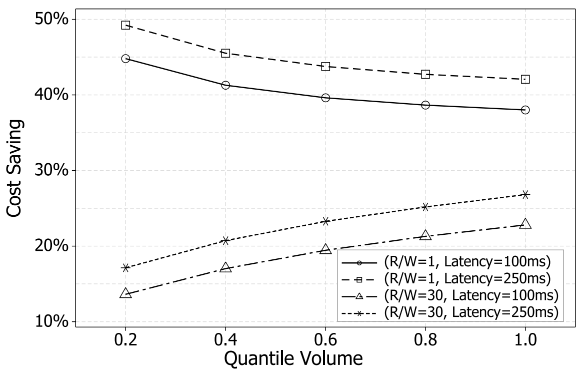


Fig. 5. Cost saving of closest-based policy vs. quantile volume.

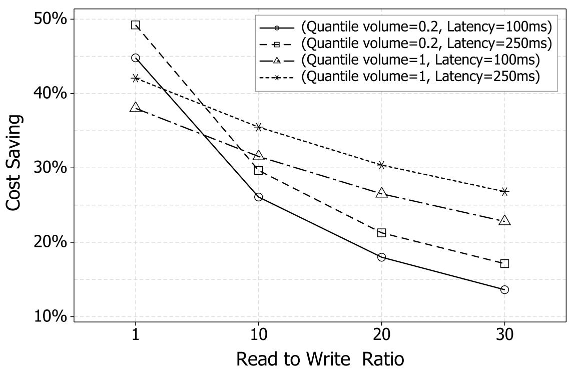


Fig. 7. Cost saving of closest-based policy vs. read to write ratio.

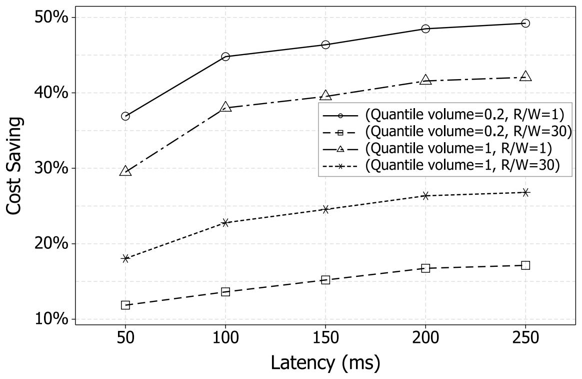


Fig. 6. Cost saving of closest-based policy vs. latency.

latency constraints. This is because that as the value of quantile volume increases, the effect of read cost reduces and storage cost becomes more dominant.

Fig. 6 illustrates the effect of latency on the cost savings. As the latency increase from 50 ms to 250 ms, as expected more improvements are observed in the cost savings for all default values of quantile volume and read to write ratio. This implies that there is a wider selection of DCs available with lower cost in storage and network resources for optimization under loose latency in comparison to tight latency.

Fig. 7 plots the effect of read to write ratio, varying from 1 (write-intensive objects) to 30 (read-intensive objects), on the cost savings. As the value of the ratio increases, the cost saving decreases. Under both latency constraints, the results show a 66% reduction in cost savings for quantile volume = 0.2, and correspondingly at most a 42% reduction for quantile volume = 1. This implies that the proposed solution is more cost-effective for write-intensive objects in comparison to the read-intensive objects due to efficient utilization of network resources for Puts.

6. Empirical studies in latency evaluation

We implemented a prototype system to provide data access management across Amazon Web Service (AWS) and Microsoft Azure cloud providers. For this purpose, we use JAVA-based AWS S3[7] and Microsoft Azure[8] storage REST APIs. With this prototype, an individual end-user can manage data across two well-known cloud providers, and measure the perceived latency for data migration.

6.1. Data access management modules

Our prototype system provides a set of modules that facilities users to store, retrieve, delete, migrate, list data across AWS and

7 Amazon S3 REST API: http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html.

8 Azure storage REST API: https://docs.microsoft.com/en-us/rest/api/storageservices/fileservices/azure-storage-services-rest-api-reference.

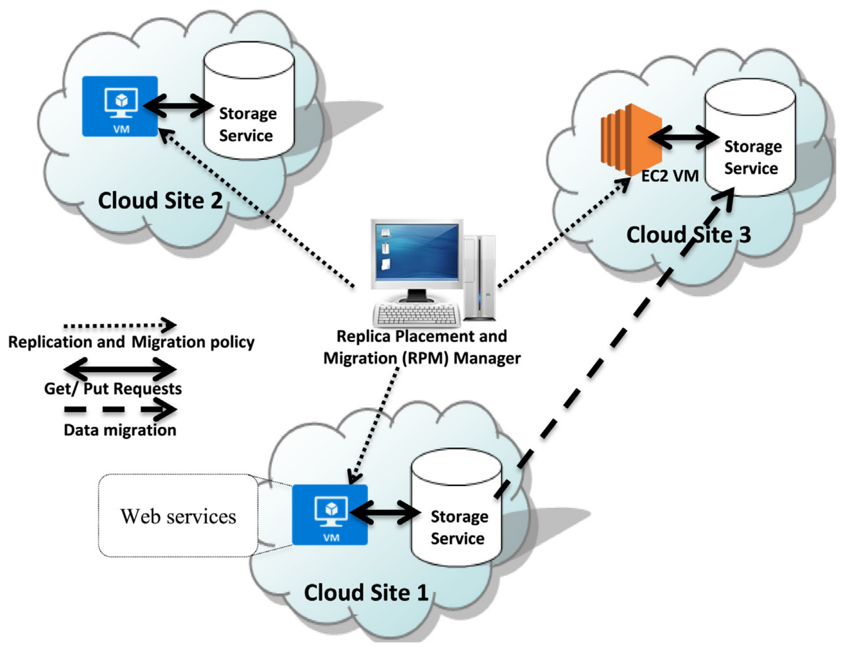


Fig. 8. An overview of prototype.

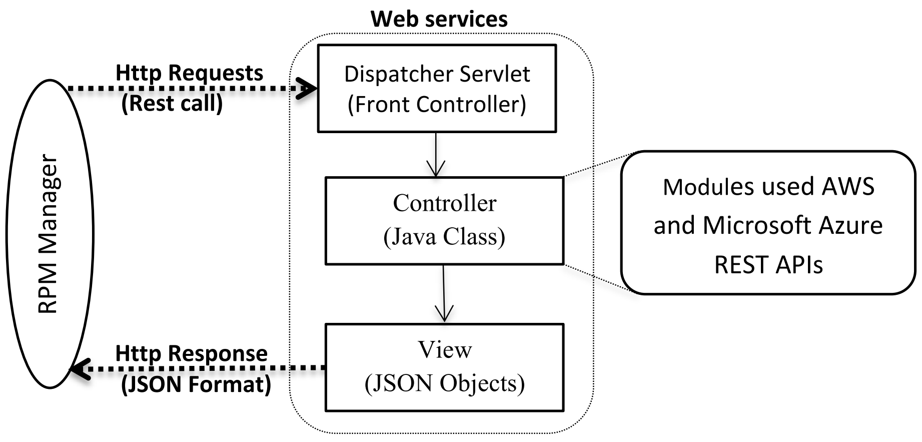

Web services
Http Requests (Rest call)
Dispatcher Servlet (Front Controller)
Controller (Java Class)
Modules used AWS and Microsoft Azure REST APIs
View (JSON Objects)
RPM Manager
Http Response (JSON Format)

Fig. 9. Web services components used in the prototype.

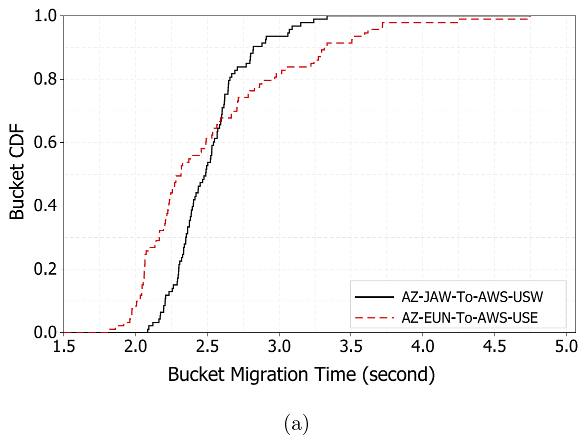
Microsoft Azure data stores [13].[9] All these services are RESTful web services that utilize AWS S3 and Microsoft Azure storage APIs in Java. They produce response in the JavaScript Object Notations (JSON) format in successful cases and error message in the error cases. We use JSON format because it is a lightweight data-interchange format and easy to understand. We discuss the provided web services in more detail in [13].

6.2. Measurement of data migration time

We design a simple prototype as shown in Fig. 8. The way in which the deployed virtual machines (VMs) should serve Puts and Gets for each object is dictated by a central replica placement and migration (RPM) manager. The RPM manager makes decision on replica placement and migration across data stores based on the proposed heuristic solution. The RPM issues Http requests (REST call) to the VMs deployed in cloud sites and receives Http responses (JSON objects). The VMs process the received requests via the deployed web services that are implemented based on Spring Model-View-Controller (MVC) framework [10] as illustrated in Fig. 9.

To measure the time spent on data migration across DCs, we utilize the federation of cloud sites from Microsoft Azure and Amazon in our prototype. We span our prototype across 3 Microsoft Azure cloud sites in `Japan West`, `North Europe`, and `South Central US` regions and 3 Amazon cloud sites in `US East (North Virginia)`, `US West(Oregon)`, and `US West (North California)` regions. In each Azure cloud site, we create a `Container` and deploy a `DS3_V2 Standard` VM instance. In each Amazon cloud site, we create a `Bucket` and deploy a `t2.medium` VM instance. All VM instances used in the prototype run Ubuntu 16.04 LTS as operating system.

[9] Data management across Amazon and Microsoft Azure. https://github.com/ymansouri/AmazonAzurePrototype.



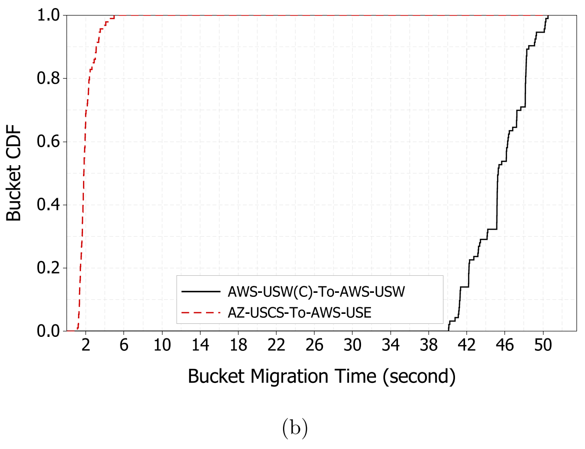


Fig. 10. CDF of data migration time (a) across regions and (b) within US region.

After the set-up, we run the heuristic algorithm for 100 users (in the Twitter traces) who are assigned to the aforementioned cloud sites. According to the replication and migration policy dictated by the heuristic algorithm, the data are stored in data stores and are integrated in a folder (analogous to bucket in Spanner [5]) for each user. When data migration happens we record the time of data transfer from source cloud site to the destination cloud site.

Fig. 10 shows the CDFs of data migration time observed for 100 buckets (each user is associated to a bucket), each of which with the size of about 47.35 MB in average. Fig. 10(a) depicts that data migration can be transmitted in several seconds across regions. About 60% of buckets are transmitted in 2.5 s from Azure DC in Japan west (`AZ-JAW`) to Amazon DC in US west (`AWS-USW`) as well as from Azure DC in Europe north (`AZ-EUN`) to Amazon DC in US east (`AWS-USE`). Also, all buckets are transmitted in 3.5 s from Asia region to US region and likewise 4.5 s from Europe region to US region. Fig. 10(b) illustrates the data migration time within US region. About 80% of buckets are migrated from Azure DC in US center south (`AZ-USCS`) to Amazon US east (`AWS-USE`) below 2 s. In contrast, bucket migration time between Amazon DC in US west (`North California`) (`AWS-USW(C)`) to another DC in US west (`Oregon`) (`AWS-USW`) is between 40–48 s for about 80% of buckets. From the results, we conclude that the duration of buckets migration is considerably low. In the case of a large number of buckets, we can transfer data in bulk across DCs with the help of services such as AWS Snowball,[10] and Microsoft Migration Accelerator.[11]

7. Conclusions and future directions

We studied the problem of optimizing the monetary cost spent on the storage services when data-intensive applications with time-varying workloads are deployed across data stores with several storage classes. We formulated this optimization problem and proposed the optimal algorithm. Since high time complexity is one of the weaknesses of this optimal algorithm, we proposed a new heuristic solution formulated as a Set Covering problem with three polices. This solution takes advantages of pricing differences across cloud providers and the status of objects that changes from hot-spot to cold-spot during their lifetime and vice versa. We evaluated the effectiveness of the proposed solution in terms of cost saving via trace-driven simulation using CloudSim simulator and real-world traces from Twitter. The evaluation results demonstrate that our solution is capable of reducing the cost of data storage management by approximately two times in some cases when compared to the widely used benchmark algorithm in which the data are stored in the closest data store to the users who access them. We also developed a prototype system to empirically measure the data migration within and regions that host DCs owned by Amazon Web Service (AWS) and Microsoft Azure cloud services. The results show that the incurred latency to transfer data between DCs is within a few seconds and is tolerable for users.

Our work can be extended in several research directions. First, since data-intensive applications experience different access pattern on their data, new algorithms to optimize cost for objects with more than two statuses (e.g., cold, warm and hot) needs to be explored and developed. Second, the home DC can be selected based on the mobility of the users. This selection criterion effects on the response time of the Gets/Puts issuing by client DCs. Third, using a quorum-based model for data consistency provides stronger consistency semantic compared to the eventual consistency as guaranteed in our work. Fourth, to determine the gap between the proposed optimal and heuristic algorithms in cost performance, it is required to compute the *competitive ratio* defined as the ratio between the worst incurred cost by the heuristic algorithm and the cost incurred by the optimal algorithm. For this purpose, we need to compute the upper-bound cost for the optimization of replicas migration (Algorithm 2) and replica placement based on covered load volume (Algorithm 3).

References

[1] D. Beaver, S. Kumar, H.C. Li, J. Sobel, P. Vajgel, Finding a needle in haystack: Facebook's photo storage, in: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 47–60.

[2] R.N. Calheiros, R. Ranjan, A. Beloglazov, C.A.F. De Rose, R. Buyya, CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, Softw. Pract. Exp. 41 (1) (2011) 23–50.

[3] F. Chen, K. Guo, J. Lin, T. La Porta, Intra-cloud lightning: Building CDNs in the cloud, in Proceedings IEEE INFOCOM, 2012, pp. 433–441.

[4] V. Chvatal, A greedy heuristic for the set-covering problem, Math. Oper. Res. 4 (3) (1979) 233–235.

[5] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, D. Woodford, Spanner: Google's globally distributed database, ACM Trans. Comput. Syst. 31 (2013) 8:1–8:22.

[10] AWS Snowball. https://aws.amazon.com/snowball/.

[11] Microsoft Migration Accelerator. https://azure.microsoft.com/en-au/blog/introducing-microsoft-migration-accelerator/.

[6] S. Das, D. Agrawal, A. El Abbadi, ElasTraS: An elastic transactional data store in the cloud, in: Proceedings of the Conference on Hot Topics in Cloud Computing, HotCloud'09, USENIX Association, Berkeley, CA, USA, 2009.
[7] S. Das, S. Nishimura, D. Agrawal, A. El Abbadi, Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration, Proc. VLDB Endow. 4 (2011) 494–505.
[8] A.J. Elmore, S. Das, D. Agrawal, A. El Abbadi, Zephyr: Live migration in shared nothing databases for elastic cloud platforms, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD '11, ACM, New York, NY, USA, 2011, pp. 301–312.
[9] L. Jiao, J. Li, T. Xu, W. Du, X. Fu, Optimizing cost for online social networks on geo-distributed clouds, IEEE/ACM Trans. Netw. 24 (1) (2016) 99–112.
[10] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, D. Kopylenko, Professional Java Development with the Spring Framework, Wrox Press Ltd., Birmingham, UK, 2005.
[11] R. Li, S. Wang, H. Deng, R. Wang, K.C.-C. Chang, Towards social user profiling: unified and discriminative influence model for inferring home locations, in The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012, pp. 1023–1031.
[12] G. Liu, H. Shen, Harnessing the power of multiple cloud service providers: An economical and SLA-guaranteed cloud storage service, in: IEEE 35th International Conference on Distributed Computing Systems, 2015, pp. 738–739.
[13] Y. Mansouri, Brokering Algorithms for Data Replication and Migration Across Cloud-Based Data Stores (Ph.D thesis), The University of Melbourne, Australia, 2017.
[14] Y. Mansouri, R. Buyya, To move or not to move: Cost optimization in a dual cloud-based storage architecture, J. Netw. Comput. Appl. 75 (2016) 223–235.
[15] Y. Mansouri, A.N. Toosi, R. Buyya, Cost optimization for dynamic replication and migration of data in cloud data centers, IEEE Trans. Cloud Comput. (2017).
[16] A. Mseddi, M.A. Salahuddin, M.F. Zhani, H. Elbiaze, R.H. Glitho, On optimizing replica migration in distributed cloud storage systems, in Proceedings of the 4th IEEE International Conference on Cloud Networking, CloudNet, Niagara Falls, ON, Canada, October 5–7, 2015, pp. 191–197.
[17] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, S. Kumar, f4: Facebook's warm BLOB storage system, in: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 14, USENIX Association, Broomfield, CO, 2014, pp. 383–398.
[18] K.P. Puttaswamy, T. Nandagopal, M. Kodialam, Frugal storage for cloud file systems, in: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12, ACM, New York, NY, USA, 2012, pp. 71–84.
[19] X. Qiu, H. Li, C. Wu, Z. Li, F.C.M. Lau, Cost-minimizing dynamic migration of content distribution services into hybrid clouds, IEEE Trans. Parallel Distrib. Syst. 26 (2015) 3330–3345.
[20] N. Tran, M.K. Aguilera, M. Balakrishnan, Online migration for geo-distributed storage systems, in: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11, USENIX Association, Berkeley, CA, USA, 2011, pp. 15–15.
[21] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, H.V. Madhyastha, SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP'13, ACM, New York, NY, USA, 2013, pp. 292–308.
[22] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, F.C.M. Lau, Scaling social media applications into geo-distributed clouds, IEEE/ACM Trans. Netw. 23 (2015) 689–702.

Yaser Mansouri is a Ph.D. student at Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, the University of Melbourne, Australia. Yaser was awarded International Postgraduate Research Scholarship (IPRS) and Australian Postgraduate Award (APA) supporting his Ph.D. studies. He received his B.Sc. degree from Shahid Beheshti University of Tehran and his M.Sc. degree from Ferdowsi University of Mashhad, Iran in Computer Science and Software Engineering. His research interests cover the broad area of Distributed Systems, with special emphasis on data management in cloud-based storage services.

Dr. Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He has authored over 625 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h- index = 123, 79, 000+citations). Software technologies for Cloud computing developed under his leadership have gained rapid acceptance and are in use at several academic institutions and commercial enterprises in 40 countries around the world. Dr. Buyya is recognized as a "Web of Science Highly Cited Researcher" both in 2016 and 2017 by Thomson Reuters, a Fellow of IEEE, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to Cloud computing. For further information on Dr.Buyya, please visit his cyberhome: www.buyya.com.